Chapter 8 - Software Reliability Measurement Experience

In this chapter, we describe a recent study of software reliability measurement methods that was conducted at the Jet Propulsion Laboratory. The first section of the chapter, section 8.1, summarizes the study, characterizes the participating projects, describes the available data, and summarizes the study's results.

The second section, 8.2, gives details on defining and collecting failure data as well as selecting software reliability measurement tools. In the JPL study, we found that one of the most important parts of a software reliability measurement program is identifying the data to be collected and setting up accurate data collection mechanisms.

Section 8.3 presents the results of the study, comparing the predictions made by the various models and drawing conclusions from the comparisons. Section 8.4 discusses one particular finding of the study in more detail - we found that a linear combination of model results tends to yield more accurate predictions than the individual models themselves over the data sets we examined.

Section 8.5 summarizes the main points of the chapter, anti suggests areas for further study.

8.1 A Practical Study at JPL

In October, 1989, the Jet Propulsion Laboratory (JPL) Directorts Discretionary Fund funded a proposal to study the ways in which software reliability measurement techniques might be applied to JPL and other NASA software development efforts. This effort continued over the next two years, analyzing failure data from previous and current JPL projects. The study resulted in a set of recommendations for the application of software reliability measurement techniques, which will be discussed in this chapter.

8.1.1 Purpose of Study

There were 4 major goals in the JPL study. These were:

1. Assess the impact of software components on the overall system reliability in the JPL environment. For representative JPL-develoed systems, we wanted to have a idea of the extent to which the software components' reliability would affect the overall system reliability. The following example shows the extent to which the software's reliability may affect the overall system reliability for one particular type of JPL system.

Figure 1 shows a system with the following software characteristics:

1. Heterogeneous distributed operating system
2. 5648 programs
3. 324,500 lines of high-level. language source code

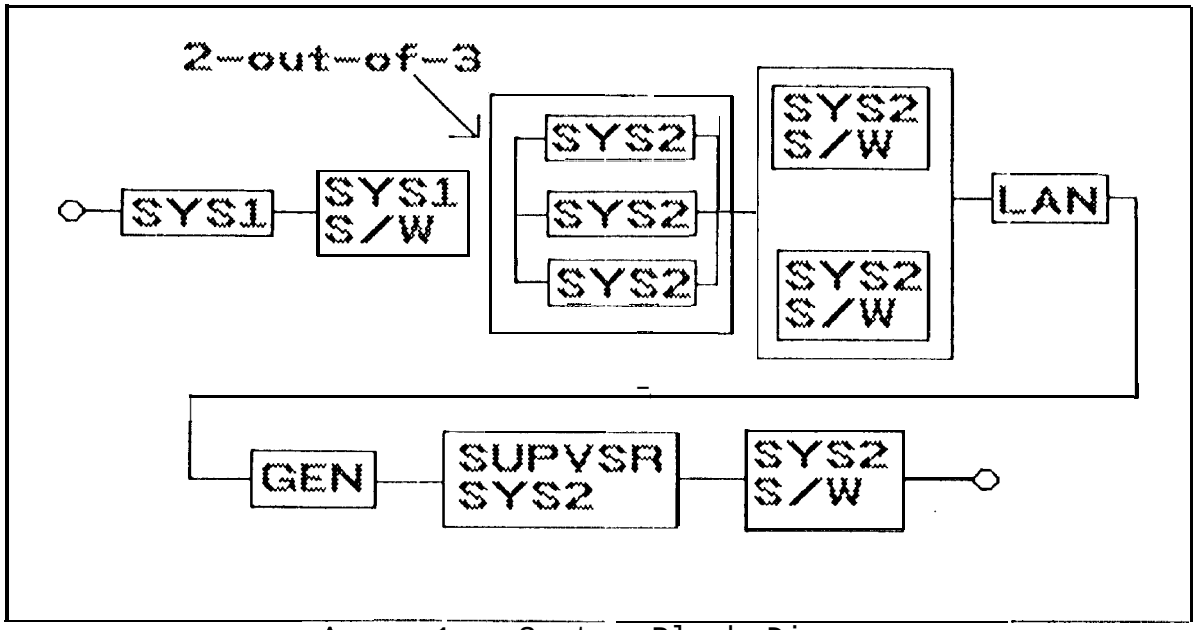

Figure 1 - System Block Diagram

The MTBFs for each hardware system component. are:

SYS1 hardware - 280 hours
SYS2 hardware - 387 hours

The SYS2 redundancy block in Figure 1 represents a 2-out-of-3 structure, yielding a MTBF of 5000 hours for that block's hardware.

LAN '---------- 10,000 hours
GEN '---------- 600 hours
Supervisory SYS2 hardware - 387 hours

If the reliability of SYS1 and SYS2 software. is assumed to be 1, the estimated MTBF for the system is 123.1 hours.

However, the methods described in section 2.3.2.1 of this handbook and [RADC87] can be used to estimate the reliability of the software components. These techniques use measurements of the software and the development

process to estimate the error density that would be seen at the end of the implementation phase, prior to any actual testing that would occur. Making assumptions about the usage patterns of the software and using information about the speed of the computers, this error density can be used to estimate the failure rate that would be seen at the start of system test. Using these methods, the estimated MTBF for the software was ,11.9 minutes. This would translate into an estimated MTBF for the system of about 11.9 minutes, a significant departure from the 123.1 hours initially estimated.

This system is typical of many current systems in that most of its functionality is implemented in the software components. There are two observations that can be made at this point. The first is that in systems of the type shown in this example, the reliability of the software is the dominant factor in determining overall system reliability. Therefore, a reliability of 1 for the software component of the system should not be assumed in estimating system reliability. Failure to take the software reliability into account will result in grossly inaccurate estimates and predictions of the system's failure behavior. The second observation, related to the first, is that for a given system, the reliability of the hardware components tends to be high quite early in the system development life cycle. The reliability of the software components, however, only increases late in the life cycle during the testing phases, as faults in the those components are detected and removed.

2. Develop quantitative software reliability models for JPL. During this study, we wanted to look at currently-available software reliability measurement techniques and identify those that would be applicable to JPL software development efforts. Fortunately, there has been a great deal of research in this area over the past 20 years, so there was no lack of information in this area. The methods we looked at can be roughly categorized as being either execution-based or early prediction. The execution based techniques are those that can be used only after the software has been executed and failures have been observed. These methods tend to be used starting with subsystem integration and continuing through system integration, acceptance test, and operations. Early prediction methods attempt to predict the reliability of the software prior to the start of the test phase, using measurements of product and development process characteristics. [RADC87] is an example of such a method. At the time of the study, almost all of the available methods were execution-based.

3. Refine and validate models using JPL software failure data. After identifying appropriate software reliability measurement techniques for JPL software development efforts, we wanted to validate and refine these techniques. The major refinement was to devise a method of combining model results that tended to yield more accurate predictions than the components of the combination.

4. Develop guidelines for the application of software reliability modeling techniques. During the study, we developed a set of guidelines that could be used by software project managers, line management, and development personnel to specify and implement a software reliability measurement program. These guidelines, available in the form of a JPL publication [JPL91], cover the following points:

   a. Establishing software reliability objectives.
   b. Preliminary software reliability model selection.
   c. Setting up data collection mechanisms.
   d. Choosing software reliability measurement tools.
   e. Final software reliability model selection.
   f. Model application and application issues.

   The guidebook also provides brief descriptions of some of the more widely-used software reliability models, describes the benefits of using these modeling techniques, and also discusses their limitations.

8.1.2 Project Selection and Characterization

For this study, we decided to look at failure data from previous and current JPL flight projects. We also analyzed failure data from a JPL-developed ground-based system for tracking and acquiring data from Earth resources satellites in high-inclination orbits. Finally, we analyzed previously-published failure data [MUSA80] for ground-based systems. This variety of data would give us a chance to see whether the reliability measurement techniques developed for one type of development effort would work well for another.

Each of the flight projects developed a planetary exploration spacecraft, along with the ground support software. For each spacecraft, the flight software was about 15,000 source lines of code. In most cases, the software was split between two processors - one to control the spacecraft attitude, and the other to process uplinked commands and relay science and engineering telemetry back to the mission operators. For the flight projects, most of the software was written in assembly language.

The resources satellites tracking and data acquisition system contained about 100,000 source lines of code, and was written in a mixture of C, FORTRAN, and various database query languages.

8.1.3 Characterization of Available Data

For all of the JPL efforts, the following data was available:

1. Date on which a failure occurred.
2. Failure description.
3. Recommended corrective action.
4. Corrective action taken.
5. Date on which failure report was closed.

For each of the flight projects, the severity of each failure was also available.

We can see right away that not quite enough information is available to apply software reliability models. Recall that to use software reliability models, data in the form of failure counts and test interval lengths, or time between successive failures is needed. The number of failures could certainly be counted, but. test interval lengths had not been systematically recorded. At. this point, it was necessary to interview members of the various development teams to get some idea of what the staffing profile looked like during the testing phases. In some cases we were able to get the necessary information from the development staff - in one specific case, one of the investigators for this study had previously been a member of the development staff for one of the projects during software integration, and was able to recall enough about the testing phase to provide fairly accurate information about the length of each test interval. For about half the efforts we studied, however, we were unable to get reliable information about test interval lengths. In these cases, we had use our knowledge of other, similar development efforts to make estimates of test interval lengths. This situation influenced the content of the guidebook, which contains a fairly detailed set of recommendations about collecting failure data.

8.1.4 Summary of Results

One of the most important things we discovered is that for the failure data we analyzed, no one model was consistently the best. It was frequently the case that a model that had performed well for one set of JPL failure data would perform badly for a different. set. We therefore recommended that for any development effort, several models, each making different assumptions about the testing and debugging process, be simultaneously be applied to the failure data. We also recommended that each model's applicability to the failure data be continuously monitored. Traditional goodness-of-.

fit tests, such as the Chi-Square or Kolmogorov-Smirnov tests, can be used. In addition, the methods described in [GHALY86] are also strongly recommended.

Another discovery was that of the software development efforts we studied, only one specified a set of reliability requirements that we felt were measurable. Strictly speaking, it is not necessary to have a reliability requirement for system in order to apply software reliability measurement techniques. It is quite possible to measure a software system's reliability during test and make predictions of future behavior. However, the existence of a requirement is very helpful in that;

1. Specifying a reliability requirement helps the users and developers focus on the components of the system that. will have the most effect on the system's overall. reliability. Potentially unreliable components can be respecified or redesigned to increase their reliability.

2. A reliability requirement will serve as a goal to be achieved during the development effort. During the testing phases, software developers and managers can estimate software reliability and determine how close they are to the required value. The difference between current and required reliability can be converted into estimates of the time and resources that will be required to achieve the goal.

We also discovered that one of the most important. aspects of setting up a software reliability measurement program is identifying the data to be collected and setting up a data collection mechanism. We found that development organizations generally have the capability to collect the type of data that is required to use software reliability measurement techniques. Every software development effort that we looked at has a mechanism for recording and tracking failures that are observed during the testing phases and during operations. At the time of the study, most projects of which we are aware also had requirements for the test staff to keep an activity log during the testing phases. Properly used, these data collection mechanisms would provide accurate failure data in a form that could easily be used by many currently-available software reliability models. However, since many software managers and developers are not aware of the types of analysis that can be done with this data, they do not devote the effort required to ensure that the collected data is complete and accurate.

Finally, we discovered that a properly-defined linear combination of model results produced more accurate predictions over the set of failure data that we analyzed than any one individual model. Of the various methods of forming combinations, we found that one in which all components of the combination are

given equal weight produced surprisingly good results [LYU91a]. Other methods in which weights are dynamically assigned to components of the combination require more computation, but produce better results than the statically-weighted method [LYU92].

8.2 Reliability Requirement Specification, Failure Data Collection, and Model Application

8.2.1 Establishing Software Reliability Requirements

Software reliability modeling techniques are used to predict a software system's failure behavior during test and operations. Software reliability requirements are specified during earlier development phases, and these modeling techniques are used to estimate the resources that will be required to achieve those requirements. The resource requirements are translated into testing schedules and budgets. Resource estimates are compared to the resources actually available to make quantitative, rather than qualitative, statements concerning achievement of the reliability requirements.

8.2.1.1 Expressing Software Reliability

Reliability and reliability-related requirements can be expressed in one of the three following ways:

1. Probability of failure-free operation over a specified time interval.
2* Mean time to failure (MTTF).
3* Expected number of failures per unit time interval (failure intensity).

The first form, the accepted definition of software reliability, is a probabilistic statement concerning the software's failure behavior. The other two forms can be considered to be related to reliability. Reliability and reliability-related requirements must be stated in quantitative terms. Otherwise, it will not be possible to determine whether the requirements have been met. To help in understanding how to develop these requirements, examples of testable and untestable reliability requirements are given in the following paragraphs.

The following statements, paraphrased from a JPL software development effort, represent a requirement for which software reliability modeling techniques can be used to determine the degree to which that requirement has been met. "Reliability quantifies the ability of the system to perform a required function under the stated conditions for a period of time. Reliability is measured by the mean time between failures of a critical component. Under the

expected operational conditions, documented in paragraph [TBD] of this requirements document, the probability of the MTTF for the software being greater than or equal to 720 hours shall be 90%."

The above requirement is stated in a testable manner. If the expected operational conditions are stated in terms of the operational hardware configuration and the fraction of time each major functional area is expected to be used (the operational profile), the test staff can then design tests to simulate expected usage patterns and use reliability estimates made during these tests to predict operational reliability.

Confidence bounds should be associated with reliability or reliability-related requirements. If the above MTTF requirement had been stated as being simply 720 hours, it would have been possible to meet that requirement with a very wide confidence interval (e.g. 90% probability of the MTTF lying between 200 and 1240 hours). This could have resulted in the delivery of operational software whose MTTF was considerably less than the intended 720 hours. Yet, the end users of the delivered software would believe that the reliability requirement had been met. Not. until the software was actually operated would the users realize the discrepancy. To avoid this problem, express the reliability requirement as the minimum value of the confidence interval. This will allow the end users to know the probability of the software meeting its reliability requirement, and permit them to plan accordingly.

An example of an untestable reliability-related requirement is now given. Again, the text is paraphrased from that found in a JPI, development effort's system requirements document. "The system is designed to degrade gracefully in case of failures. System fault. protection shall ensure that no error or component failures will compromise as a first priority, [safety restriction], and as a second priority, minimum mission science objectives stated in paragraph [TBD]. Accordingly, each instrument shall be designed so that if one fails (either through hardware or software failures), it will not jeopardize the safety of the system or damage adjacent instruments. This includes provision for isolation from the system via the instrument power supply. If a system fault occurs, the system will automatically stop any science data gathering and go to a safe state. After a safe state is achieved and subsystems are re-initialized, science can be resumed."

The foregoing requirement does not provide a basis against. which the failure behavior of the system under development can be measured, as it contains no quantitative statements concerning the system's failure behavior. Rather, it is a statement of design constraints that are intended to localize damage resulting from a component failure to the immediate area (e.g. assembly, subsystem) in which the failure occurred. During subsequent phases of system development, it is indeed possible to determine whether such

constraints have been reflected in the system design and implementation. However, this information alone is not sufficient to make quantitative statements concerning the system's failure behavior. Although specifying constraints such as these is an important aspect of system specification, specific reliability requirements, similar in form to the first reliability requirement discussed in this section, would have to be provided if it were intended to use reliability estimation techniques to determine compliance to a reliability requirement.

8.2.1.2 Specifying Reliability Requirements

To specify reliability requirements, use one or more of the three methods described below. The methods are:

1. System balance.
2. Release date.
3. Life cycle cost optimization.

It is possible to use one of these methods for developing the requirements for one component of the system, and another for a separate component.

The system balance method is primarily used to allocate reliabilities among components of a system based on the overall reliability requirement for that system. The basic principle of this method is to balance the difficulty of development work on different components of the system. The components having the most severe functional requirements *or* are the. most technologically advanced are assigned less stringent reliability requirements. In this way, the overall reliability requirement for the system is met while minimizing the effort required to implement the most complex components. For software, this might translate to assigning less stringent reliability requirements to functions never before implemented or functions based on untried algorithms. This approach generally leads to the least costly development effort in the minimum time. The system balance method is frequently used in developing military systems.

The second approach is used when the release date is particularly critical. This is appropriate for flight systems facing a fixed launch window. The release date is kept fixed in this approach. The reliability requirement is either established by available resources and funds, or is traded off against these items. With this approach, it is desirable to know how failure intensity trades off with release date. First, the way in which the failure intensity trades off with software execution time is determined. This execution time is then converted to calendar time. The following example uses the Goel-Okumoto and Muss-Okumoto models.

For the Goel-Okumoto model, the relationship between the ratio of failure intensity change during test and the execution time is given by:

$$\tau = \frac{1}{b} \ln \frac{\lambda_0}{\lambda_P}$$

τ = elapsed execution time
λ_0 = initial failure intensity
λ_P = required failure intensity
b = failure intensity decay parameter

For the Muss-Okumoto model, this relationship is given by:

$$\tau = \frac{1}{\theta \lambda_0} \left(\frac{\lambda_0}{\lambda_P} - 1 \right)$$

τ, λ_0, and λ_P as above
θ = failure intensity decay parameter

For this example, the failure history data from one of the testing phases of a JPL flight program is used. Applying the Goel-Okumoto and Muss-Okumoto models to this data, the following model parameter and failure intensity estimates are obtained:

Goel-Okumoto	Muss-Okumoto
λ_0 = .4403 failures/ CPU hour a = 202.52 failures b = 6.044 x 10^{-7} per failure	λ_0 = .5064 failures/ CPU hour θ = 1.693x10^{-6}/failure

The above equations can be used to determine the amount of test time that will be needed for various failure intensity improvement factors:

Failure Intensity Improvement Factor λ_0/λ_P	Execution Time (CPU Hours)	
	Goel-Okumoto	Muss-Okumoto
10	1058	2916
100	2118	32076
1000	3177	323,677
10000	4236	3,239,690

The relationship of execution time to calendar time will vary with the test phase, development methods, and the type of software under development. Knowledge of how the failure intensity ratio varies with execution time can be used to determine the general shape of the relationship between calendar time and failure intensity ratio. An example of this relationship, adapted from [MUSA87], is shown in Figure 2. This relationship can be used to determine an attainable failure intensity ratio, given the release date and available resources. The failure intensity requirement can then be obtained from this ratio.

Note the differences between the predictions made by the two models. In the Muss-Okumoto model, the relationship between additional execution time needed and the improvement factor is linear, while in the Goel-Okumoto model it is logarithmic. At this point, a choice between the two models must be made. Since it is not possible to know a priori which model is best suited to the data [6], the applicability of models to a set of failure data must. be evaluated while the models are being applied. Techniques for determining model applicability, based on those reported in [GHALY86], are summarized in paragraph 8.2.8.2. Once the model most applicable to the failure data has been identified, that. model's relationship between failure intensity" ratio and execution time can be used in conjunction with the relationship between execution and calendar time to determine the failure intensity requirement.

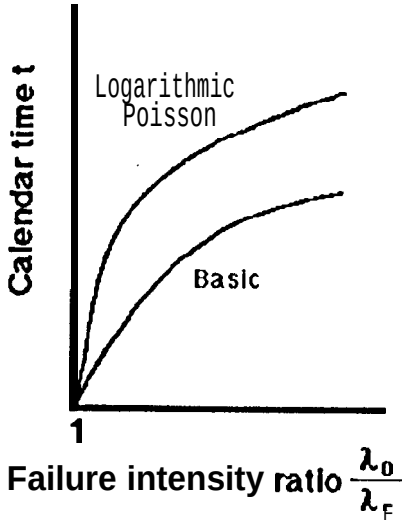

Figure 2 - Failure Intensity Ratio vs. Calendar Time

The third approach, life cycle cost optimization, is described in the following paragraphs. Although this technique would be difficult to apply to spacecraft flight software (since the launch date is usually fixed), it would work well for ground-based software not directly in the uplink or downlink path (e.g. image processing software, long-term spacecraft scheduling, ground-based spacecraft simulation) . The basis for optimization in this technique is the assumption that reliability improvement is obtained by more extensive testing. Costs and schedules for non-testing phases are assumed to be constant. The part of development cost due to testing decreases with higher failure intensity requirements , while the operational. cost increases. The total cost therefore has a minimum. This is shown below in Figure 3.

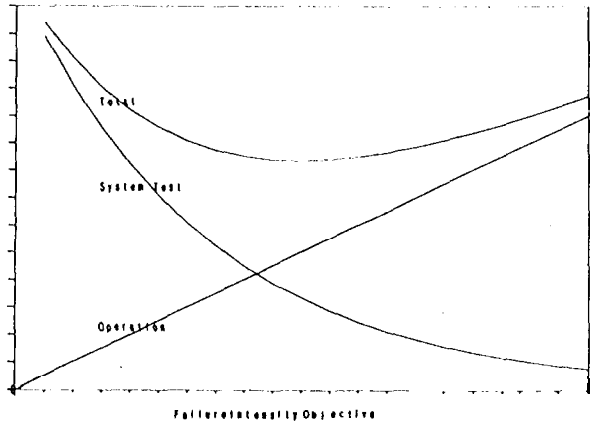

Figure 3 - Test, Operational, and Total Cost

To find this minimum, testing cost as a function of failure intensity must be computed. If testing cost can be related to calendar time, and if the relationship between calendar and execution time is known, this calculation can be done for a specific model. Similarly, the operational cost as a function of failure intensity must be computed. The following costs must be considered:

1. Terminating an improperly functioning program in an orderly manner.

2. Reconstructing affected databases.

3. Restarting the program.

4. Determining the cause(s) of the failure.

5. Developing procedures to prevent further failures of that type.

6. Repairing the fault(s) causing the failure if the severity and criticality of the failure warrants corrective action.

7. Testing the software to validate any repairs.

8. Effect of similar failures in the future on mission or program success.

Determination of the failure intensity requirement then becomes a constrained-minimum problem that can be solved analytically or numerically.

A closed form expression for the operational cost of failures is difficult, if not impossible, to obtain. Analyzing the failure history of similar historical projects would seem to be an effective way of estimating the costs, since institutional problem reporting and tracking mechanisms require that the effort required to identify and correct a problem be recorded along with the problem description. Frequently, however, this information is not recorded, nor is it necessarily accurate when it is recorded. Without a statistically significant universe of projects to analyze, determination of the operational cost of failure becomes impossible. This method for determining failure intensity requirements is not recommended unless there is a well-established, practical mechanism for systematically recording and tracking operational failure cost.

8.2.2 Setting up a Data Collection Process

When setting up a software reliability measurement program, there are several pitfalls to be avoided. First, there is often the notion that every bit of information about the program and what happens to it as it evolves over the life cycle needs to be kept. Too many organizations do not have a clearly defined objective for their data collection process. As a result, much effort is expended with little gain. It is often felt that all of the data is necessary so that if someone identifies a specific objective, the pertinent data to support it has most likely been gathered. Frequently, this approach results in a costly impact on the software development process with little or no positive impact. There have been many instances in which large data collection efforts have been implemented without any capability to analyze the data. Clearly defined objectives are necessary to help define the measurement requirements. In addition, when a large amount of data is required, it is usually the development staff that is affected.

Cost and schedule suffer because of the additional effort of collecting the data. Project management complains about the large amount of overhead involved in the data collection without any constructive feedback that could help the development process.

Use the following sequence of steps to set up a data collection process. These steps are based on work done by the AIAA Space-Based Observation Systems Committee on Standards (SBOS COS) as well as work done in this study.

1. Establish the objectives. The need for this step has just been discussed. However, the importance of doing this cannot be overemphasized. Establishing the objectives is often the distinguishing point between successful and unsuccessful data collection efforts.

2. Develop a plan for the data collection process. Involve all of the parties that will be involved in the data collection and analysis. This includes designers, coders, testers, QA staff, and line and project software managers. This insures that all parties understand what is being done and the impact it will have on their respective organizations. The planning should include the objectives for the data collection and a data collection plan. Address the following questions:

   a. How often will the data be gathered?
   b. By whom will the data be gathered?
   c. In what form will the data be gathered?
   d. How will the data be processed, and how will it be stored?
   e. How will the data collection process be monitored to insure the integrity of the data and that the objectives are being met?
   f. Can existing mechanisms be used to collect the data and meet the objectives?

3. If any tools have been identified in the collection process, their availability, maturity and useability must be assessed. Commercially available tools must not be assumed to be superior than internally developed tools. Reliability, ease-of-use, robustness, and support are factors to be evaluated together with the application requirements. If tools are to be developed internally, plan adequate resources - cost and schedule - for the development and acceptance testing of the tool.

4. Train all parties in use of the tools. The data collectors must to understand the purpose of the measurements and know explicitly what data is to be collected. Data analysts must understand a tool's analysis capabilities and limitations.

5* Perform a trial run of the data plan to iron out any problems and misconceptions. This can save a significant amount of time and effort during software development. If prototyping is being done to help specify requirements or to try out a new development method, the "trial run" data collection could be done during the prototyping effort.

6. IMPLEMENT THE PLAN.

7. Monitor the process on a regular basis to provide assurance that objectives are met and that the software is meeting the established reliability goals.

8. Evaluate the data on a regular basis. Don't make the reliability assessment after software delivery. Waiting until after delivery defeats the usefulness of software reliability modeling because you have not used the information for managing the development process. Based on the experiences reported in [LYU91], [LYU91a], and [LYU91b], weekly evaluation seems appropriate for many development efforts.

9. Provide feedback to all parties. This should be done as early as possible during data collection and analysis. It is especially important to do so at the end of the development effort. It is very important to provide feedback to those involved in data collection and. analysis so they will be aware of the impacts of their efforts. Parties who are given feedback will be more inclined to support future efforts, as they will have a sense of efficacy and personal pride in their accomplishments.

8. 2.3 Defining Data to be Collected

A significant fraction of the data required is already tracked by existing JPL data collection systems. These include the Problem/Failure Report, (P/FR), the Failure Report (FR), the Incident/Surprise/Anomaly Report (ISAR), and the Discrepancy Report (DR) systems. Collect the following information during those testing phases for which reliability estimates will be made.

1. Time between successive failures. Collect the execution time between successive failures. If execution time is unavailable, testing time between successive failures, measured by calendar time, can be used as a basis of approximation. (This may result in less accurate estimates of reliability behavior.) Collect the start and completion time of each test session. Collect the times between failures (interfailure times).

If interfailure times cannot be collected, then collect test interval lengths and the number of failures encountered during each test interval. Failure frequency information seems to be more easily collected than interfailure times. Test interval lengths must be accurately recorded for usable estimates to be made with this type of data. Also collect the CPU utilization during the test periods to determine the relationship between CPU and calendar time.

For many development efforts, failure frequency information is the only available type. However, some software reliability tools can use only interfailure times as input. In this instance, the failure frequency data can be transformed to time-between-failures data in one of two ways. 'rhe first way is to randomly allocate the failures over the length of the time interval. According to [MUSA87], for many models this randomization will result in estimation errors of less than 15%. A second way, easier to implement, is to allocate the failures uniformly over the interval length. For example, if an interval is three hours in length and 3 failures occurred during that interval, the time between successive failures would then be one hour.

Recall that the way in which uncertainty in the reported failure times affects the accuracy of modeling results. Problem reporting mechanisms should be structured such that the mechanism's resolution is greater than the average interfailure time throughout the test cycle.

2. Functional area tested during each test interval. This can be done with reference to a software requirements document or a software build plan. To illustrate the importance of tracking this information, reliability estimates made if this information is tracked are compared below to reliability estimates using the same failure data but not tracking the functional areas tested. Failure data from the software integration testing phase of subsystem from a previous JPL development effort was used for this example. The software reliability estimates were made using the public-domain software reliability modeling tool SMERFS. The Goel-Okumoto NHPP model was applied to the data. The software was assumed to be composed of two largely independent functional areas, and that each functional

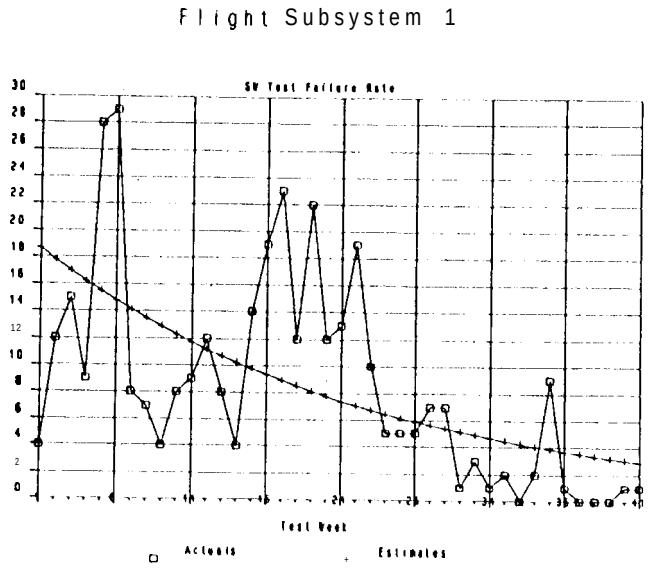

Figure 4 - Application of Goel-Okumoto Model to Entire Data Set

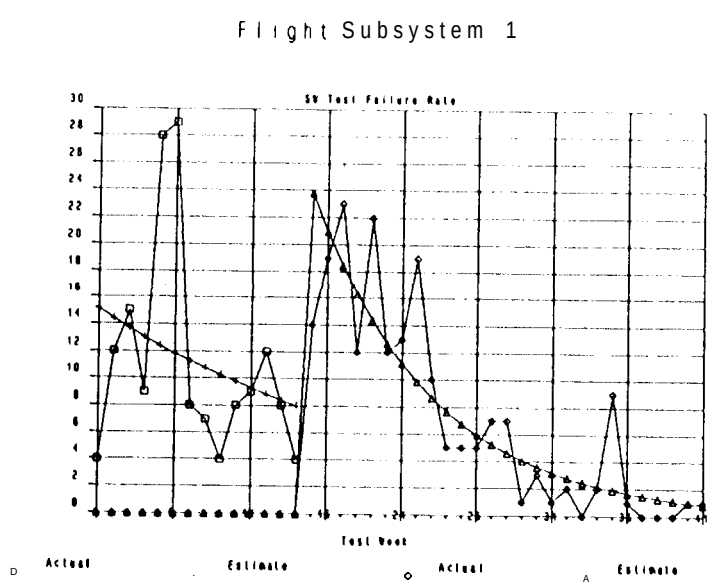

Figure 5 - Goel-Okumoto Model - Separate Modeling of Distinct Functional Areas

area would be executed 50% of the time during operations. In producing the estimates seen in Figure 4, the model was applied to the entire set of failure data. This yields an estimated failure rate of three failures per week at week 41 of the testing phase.

The actual failure rate curve, however, is bimodal. There is clearly a change in the test procedure after week 14 of the testing phase. If it is known that the software is composed of two distinct functional areas, and that after week 14, a different portion of the software is being tested than during the first 14 weeks, the reliabilities of the two functional areas can be separately modeled to yield a more accurate reliability estimate.

Figure 5 shows the reliability estimates for the individual functional areas. By the end of week 14, the expected number of failures per week is 8 f-or the first functional area. During the interval between weeks 15 and 41, only the second functional area is tested. By the end of week 41., the expected number of failures per week is 1. If the software is delivered to operations at the end of week 41, and assuming that the functional areas are executed with equal frequency during each week of testing, it is seen that during operations, 4 errors per week can be expected while executing the first functional area, and .5 errors per week can be attributed to the second functional area. The resulting estimate of 4.5 errors per week is significantly different from the 3 errors per week that were estimated without taking the change in test focus into account.

3. Significant events that may affect the failure behavior during test:

   a. Addition of functionality to the software under test or significant modification of existing functionality. If the software under test is still evolving, the failure intensity may be underestimated during the early stages of the program's development, yielding overly optimistic estimates of its reliability,

   b. Increases or decreases in the number of testers. This will increase or reduce the failure frequency (expressed in calendar time) as testers are added or taken away from the development effort. The time spent by each tester in exercising the software must be recorded so that the failure frequency or times-between-failures inputs to the models are accurate,

c. Changes in the test environment (addition/removal of test equipment, modification of test equipment) . If the test equipment is modified during a test phase to provide greater throughput, the interfailure times and failure frequencies recorded subsequently to the modification will have to be adjusted to be consistent with the failure data recorded prior to the modification. For instance, if the clock speed in the test computer is increased by a factor of two, the test intervals subsequent to the clock speed increase will need to be half as long as they were prior to the speedup if failure frequency information is being recorded. If interfailure times are being recorded, the interfailure times recorded subsequent to the speedup will. have to multiplied by 2 to be consistent with the times-between-failures recorded before the speedup occurred.

d. Changes in the test. method (e.g. switching from "white box" to "black box" testing, changing the stress to which the software is subjected during test) . If the test method changes during a testing effort, or if the software is exercised in a different manner, new estimates of the software's reliability will have to be made, starting at the time when the testing method or testing stress changed.

Interfailure times expressed in terms of CPU time are the preferred data. However, failure frequency data is also recommended since existing problem reporting mechanisms can often be used. The relative ease of collecting this information will encourage the use of reliability modeling. Currently, most JPL problem reporting systems collect the number of failures per unit test time interval. If your projects have existing mechanisms for collecting software failure data during developmental testing, use this data to obtain time-between-failures or failure frequency data.

If failure frequency data is used, a useful length for the test interval must be determined. This is influenced by such considerations as the number of testers, the number of available test sites, and the relative throughputs of test sites. Many development efforts summarize their findings on a weekly basis. For many development efforts, a week during subsystem or system-level testing is a short enough period of time that the testing method will not change appreciably. For the development efforts reported on in [LYU91], [LYU91a], and [LYU91b], enough errors were found in a week's time during the early stages of test to warrant recomputing the reliability.

Many development projects require that test logs be kept during developmental and system-level testing, although the information recorded in these logs is generally not as accurate as that tracked by the problem reporting system. Used as intended, these logs can be used to increase the accuracy of the failure frequency or interfailure time data available through the problem tracking system being used. Without much effort beyond that required to record failures, the following items can be recorded:

a. Functionality being tested. The functionality can be related to items in a software build plan or requirements in a software requirements document. The reliability for each functional area should be modeled separately.

b. Test session start date and time.

c. Test session stop date and time.

In addition, it may be possible to collect CPU utilization data. from the test bench's accounting facilities for each test period. recorded.

If only one functional area is to be tested during a session, record only one start and stop time. If more than one functional area is to be tested, however, start and stop times should be recorded for each functional area. If testing is being done at more than one test site, keep a log at each test site. To determine test interval lengths, use the test logs from all test sites to determine the amount of testing time spent in a fixed amount of calendar time. Count the number of failure reports from all test sites written against that functional area in the chosen calendar interval to determine the failure counts. These failure counts and test interval lengths can then be used as inputs to the software reliability model(s). Note that the reliability of each functional area is separately determined.

8.2.4 Choosing a Preliminary Set of Software Reliability Models

After specifying the software reliability requirements, make a preliminary selection of software reliability models. Examine the assumptions that the models make about the development method and environment to determine how well they apply to the effort at hand. For instance, many models assume that the number of errors in the software has an upper bound. If software testing at the subsystem level does not occur until the software is relatively mature, and if there is a low probability of making changes to the software actually being tested, models making this assumption can be included in the preliminary selection (e.g. Goel-Okumoto model, Muss Basic model). If, on the other hand, significant changes are being made to the software at the same time it is being tested, it would be more appropriate to choose from those models that do not

assume an upper bound to the number of faults (e.g. Muss-Okumoto and Littlewood-Verrall models). Many models also assume "perfect debugging." If previous experience on similar projects indicates that most repairs do not result in new faults being inserted into the software, choose from those models making this assumption (e.g. Goel-Okumoto model, Muss-Okumoto model). However, if a significant number of repairs result in new faults being inserted into the software, it is more appropriate to choose from those models that do not assume perfect debugging (e.g. Littlewood-Verrall model).

It is important to note that there is currently no known method of evaluating these assumptions to determine a priori which model will prove optimal for a particular development effort [GHALY86]. Users are advised that this preliminary selection of models will be a qualitative, subjective evaluation. After a model has been selected, its performance during use can be quantitatively assessed [GHALY86]. However, these assessment techniques cannot be applied to the preliminary selection.

There are additional criteria by which software reliability models can be evaluated. Six model selection criteria identified in [LYU91] are reproduced below:

1. Model validity: Includes measurement accuracy for current failure intensity, prediction of the time to finish testing with associated date and costs, and prediction of the operational failure rate.

2. Ease of measuring parameters: Includes cost, schedule impact for data collection, and physical significance of parameters to software development process.

3. Quality of assumptions: Includes closeness to the real world, and adaptability to a specific development environment. This is discussed in more detail. in section 8.2.8.1, "Applicability of Model Assumptions."

4. Applicability: Includes ability to handle program evolution and change in test and operational environment.

5. Simplicity: In concept, data collection, program implementation, and validation.

6. Insensitivity to noise: Minimal response to insignificant changes in input data and parameters without losing responsiveness to significant-differences (e.g. change in test method, changing test scenarios) .

	Model	Criteria						Results
		1	2	3	4	5	6	
1	Jelinski-Moranda	y	Y	-	Y	Y	Y	Y
2	Weibull	y	n	Y	Y	n	n	n
3	Duane Model	-	Y	Y	Y	Y	Y	Y
4	Rayleigh Model	n	Y	n	-	Y	n	n
5	Shick-Wolverton	n	Y	n	n	Y	n	n
6	Muss Basic Model	y	Y	-	Y	Y	Y	Y
7	Goel-Okumoto	y	Y	-	Y	Y	Y	Y
8	Bayesian Jelinski-Moranda	y	n	Y	Y	n	Y	
9	Littlewood Model	Y	n	Y	Y	n	y	
10	Bayesian Littlewood	y	n	Y	Y	n	y	
11	Keiller-Littlewood	y	n	Y	Y	n	y	
12	Littlewood-Verrall	y	n	Y	Y	n	y	-
13	Schneidewind Model	y	Y	-	Y	y	y	y
14	Muss-Okumoto	y	Y	Y	Y	y	y	y
15	Littlewood NHPP	y	n	Y	Y	n	y	-

Table 8-1 - Results of Applying Model Evaluation Criteria to 15 Model

"Y" signifies that a particular model passes the evaluation criterion in a specific column

"n" signifies that a particular model does not pass the evaluation criterion in a specific column

"." signifies uncertainty as to whether a model passes the evaluation criterion in a column.

In the "Results" column, those models with *zero or* one "n" are marked as "y" (pass). Models with three or more "n" are marked as "n" (fail). Otherwise, models are marked as "-" (not sure).

As with the model assumptions, the preliminary evaluation of models with respect to these criteria will be qualitative and subjective. As part of the work accomplished for the JPL study, fifteen different models were evaluated with respect to these criteria. The results, presented in [LYU91], are reproduced in Table 8-1 above. Users having had no practical experience with software reliability models are advised to use the models indicated with a "y" in the "Results" column as a preliminary selection. Otherwise, users may apply the above criteria to the models they have used to make a preliminary selection.

8.2.5 Choosing Reliability Modeling Tools

Having addressed the issues of model selection and data collection, reliability modeling tools must now be selected. Information about currently-available tools and criteria for selecting them are given elsewhere in this book. For the JPL study, the public-domain reliability modeling tools SMERFS, version 4, was selected. At the time of the study, the points in its favor were:

1. It implements a large number of models (9 unique models are implemented in version 4 of this program) .

2. Inputs to this model can be in the form of time-between-failures or failure frequencies.

3. Inputs to SMERFS can be from ASCII text files, *or* the user can enter data from the keyboard. If file input is desired, the ASCII file can be created from the application tracking the failure data.

4. For many models, SMERFS allows the user to predict the number of errors that will be found in a given time interval. This can be used to predict failure rates in the future, thereby allowing the user to estimate how much more testing time will be required to achieve a specified reliability requirement.

5. In estimating model parameters using the failure data, SMERFS will provide confidence values for these parameters if maximum likelihood estimates are requested.

6. It allows the user to produce plots of actual and estimated failure behavior, display summary statistics of the failure data, modify existing failure data, and perform linear and non-linear transformations on the failure data.

7. It produces plot files that can be imported into many spreadsheet or drawing packages as ASCII text. These

files can be easily manipulated to produce failure rate and cumulative number of failures plots.

8. The printed documentation for SMERFS adequately describes the capabilities of the tool and helps the user apply the models to failure data.

9. SMERFS runs on a wide variety of platforms and operating systems. Some of these are:

   MSDOS-based machines
   The Macintosh
   UNIX workstations
   VAX workstations
   CDC Cyber

   The fact that the FORTRAN 77 source code is shipped along with the executable file allows users to compile and run the program in almost any environment. Since no extensions to FORTRAN 77 are used, SMERFS functionality should remain unchanged, regardless of the environment in which it executes.

10. The user can modify the on-line help file, since it is shipped as a text file along with the source files.

11. SMERFS is in the public domain and can be obtained free of charge by contacting Dr. William Farr of the Naval. Surface Weapons Center, Code B-10, Dahlgren, VA, 22448.

Although it had many advantages, SMERFS version 4 did have some shortcomings. The most important of these are:

1. There are currently no capabilities, such as those described in [GHALY86], to determine the applicability of a specific model to a set of failure data. Version 5 of SMERFS does have these capabilities

2. Version 4 of SMERFS does not allow the user to combine the results of different models in the manner suggested, in [LYU91], [LYU91a], and [LYU91b]. The public-domain tool CASRE, described elsewhere in this book, does have this capability. ,

3. The variety of plots was limited. Version 4 of SMERFS produced plots of actual and estimated interfailure times and failure frequencies. Reliability and cumulative number of failures (actual and estimated) would have been useful additional features, as would a display of confidence bounds on these plots. However, these types of plots are not available.

4* The graphics quality was low. SMERFS itself produces "line printer" quality plots. However, the capability of producing a plot file that can be imported by spreadsheets, statistics, and drawing packages alleviates this problem.

5. The on-line help facilities *were* limited to descriptions of the various models. On-line help could only be invoked when the user wishes to execute a model.

6. The editing facility was a simplified line editor, rather than a screen editor. In addition, the editor commands are implemented as choices on a full-screen menu, so the file being edited is not always visible. Switching between the menu and the file being edited can be disorienting.

In conclusion, SMERFS was the preferred software reliability modeling tool, providing that it was not necessary for the user to determine the applicability of specific models to a failure data set in the manner described in [GHALY86]. .

8.2.6 Final Reliability Model Selection

Having investigated model assumptions, completed the preliminary model selection, and identified tools implementing these models, the final. model selection must now be made. More than one model may be selected - there have been suggestions that the results of two *or* three models be combined in some fashion to yield better reliability estimates than those available from a single model ([GHALY86], [LYU91], [LYU91a], [LYU91b]). Results of the investigation conducted for the JPL study indicate that the Goel-Okumoto, the Muss-Okumoto, and the Littlewood-Verrall are applicable to a wide range of JPL development efforts. Over the entire set of data examined for this study, an arithmetical average of the estimates from the Goel-Okumoto, the Musa-Okumoto, and the Littlewood-Verrall models performed consistently better than the individual models ([LYU91], [LYU91a], [LYU91b]).

8.2.7 Model Application and Application Issues

After setting up a data collection mechanism and selecting the model(s) and tool(s) to support a software reliability measurement program, software reliability measurement can be started. Do not attempt to measure software reliability during unit test. Although observed errors may be recorded during this testing phase, the individual units of code are too small to make valid software reliability estimates. Experience with JPL data indicates that the earliest point in the life cycle at which meaningful software reliability measurements can be made is at the subsystem software

'integration and test level. Other organizations report similar findings. Experience gained in the JPL study, as well as empirical evidence reported in [MUSA87], indicate that software reliability measurement should not be attempted. for a software system containing fewer than 2000 lines of uncommented source code. No way of analytically determining the minimum size of a software system whose reliability can be modeled is currently known.

8.2.8 MODEL APPLICATION ISSUES

The following paragraphs deal with three major issues of applying software reliability models. First of all, each model makes assumptions about the development process. These assumptions may not be valid for specific development efforts. The most questionable assumptions are listed and the impacts they may have on the accuracy of reliability estimates are discussed.

Secondly, a priori selection of the best model for a development effort does not seem to be possible. However, it is possible to determine the applicability of a model to a particular set of failure observations after use of the model has started. Paragraph 8.2.8.2 summarizes the techniques described in [GHALY86] for determining the applicability of a model to a set of failure data, and describes ways in which the results of such an "applicability evaluation" may be used to choose an appropriate model.

Finally, software that is under test may be simultaneously undergoing change. The volume of the changes may affect the accuracy of the reliability estimates. Paragraph 8.2.8.3 discusses ways of dealing with evolving programs.

8.2.8.1 Applicability of Model Assumptions

This section explores in greater detail some of the assumptions made by some of the more widely-used software reliability models. These assumptions are made to cast. the models into a mathematically tractable form. However, there may be situations in which the assumptions for a particular model or set of models do not apply to a development effort. In the following paragraphs, specific model assumptions are listed and the effects they may have on the accuracy of reliability estimates are described.

a. During testing, the software is operated in a manner similar to the anticipated operational usage. This assumption is often made to establish a relationship between the reliability behavior during testing and the operational reliability of the software. In practice, the usage pattern during testing can vary significantly from the operational usage. For instance, functionality

that is not expected to be frequently used during operations (e.g. system fault protection) will be extensively tested to ensure that it functions as required when it is invoked.

One way of dealing this issue is to model the reliability of each functional area separately, and then use the reliability of the least reliable functional area to represent the reliability of the software system as a whole. Predictions of operational reliability that are made this way will tend to be lower than the reliability that is actually observed during operations, provided that the same inputs are used during test as are used during operations. If the inputs to the software during test are different from those during operations, there will be no relationship between the reliability observed during test and operational reliability.

b. There are a fixed number of faults contained in the software. Because the mechanisms by which errors are introduced into a program during its development are poorly understood at present, this assumption is often made to make the reliability calculations more tractable. Models making this assumption should not be applied to development efforts during which the software version being tested is simultaneously undergoing significant changes (e.g. 20% *or* more of the existing code is being changed, or the amount of code is increasing by 20% or more). Among the models making this assumption are the Jelinski-Moranda, the Goel-Okumoto, and the Muss Basic Models. However, if the major source of change to the software during test is the correction process, and if the corrections made do not significantly change the software, it is generally safe to make this assumption. In practice, this would tend to limit application of models making this assumption to subsystem-level integration or later testing phases.

c. No new errors are introduced into the code during the correction process. Although there is always the possibility of introducing new errors during debugging, many models make this assumption to simplify the reliability calculations. In many development efforts, the introduction of new errors during correction tends to be a minor effect. In [LYU91], several models making this assumption performed quite well over the data sets used for model evaluation. If the volume of software, measured in source lines of code, being changed during correction is not a significant fraction of the volume of the entire program, and if the effects of repairs tend to

be limited to the areas in which the corrections are made, it is generally safe to make this assumption.

d. Detections of errors are independent of one another. This assumption is not necessarily valid. Indeed, there is evidence that detections of errors occur in groups, and that there are some dependencies in detecting errors. The reason for this assumption is that it. enormously simplifies the estimation of model parameters. Determining the maximum likelihood estimator of a model parameter, for instance, requires the computation of a joint probability density function (pdf) involving all of the observed events. The assumption of independence allows this joint pdf to be computed as the product of the individual pdfs for each observation, keeping the computational requirements for parameter estimation within practical limits.

Practitioners using any currently-available models have no choice but to make this assumption. All of the models analyzed as part of the JPL study and reported on in [LYU91], [LYU91a], [LYU91b] make this assumption. Nevertheless, practitioners from AT&T, Hewlett Packard, and Cray Research report that the models produce fairly accurate estimates of current reliability in many situations. If inputs to the software are independent of each other and independent of the output, error detection dependencies may be reduced.

8.2.8.2 Determining Model Applicability

As previously stated, there is no known method of determining a priori the "best" reliability model for a software development effort. However, once use of a model has started, analyses can be done to determine the applicability of the model to the failure data used as input to the model. The following paragraphs summarize these analysis methods, which are detailed in [GHALY86].

The **prequential** likelihood ratio can be used to discredit one model in favor of another for a particular set of failure data. Recall that software reliability models treat the time to failure as a random variable T_i. The cumulative distribution function $F_i(t)$ for this random variable is based upon the previous i-1 observed times to failure $t_1, t_2, \ldots, t_{i-1}$. The probability density function $f_i(t)$ of the random variable T_i is the time derivative of the cumulative distribution function. For one-step ahead predictions of $T_{j+1}, T_{j+2}, \ldots, T_{j+n}$, the prequential likelihood is given by:

$$PLR_n = \prod_{i=j+1}^{\cdots} f_i(t_i)$$

A comparison of two models, A and B, may be made by forming the prequential likelihood ratio $PLR_n = PL_n^A / PL_n^B$. The reliability practitioner believes that either model A is true with p(A) or that model B is true with probability p(B) = 1 - p(A). The practitioner observes the failure behavior of the system, bakes predictions using the two models A and B, and compares the predictions to the actual behavior via the prequential likelihood ratio. When predictions have been made for $T_{j+1}, T_{j+2}, \ldots, T_{j+n}$, the PLR is given by:

$$PLR_n = \frac{p(t_{j+n}, \ldots, t_{j+1} | t_j, \ldots, t_1, A)}{p(t_{j+n}, \ldots, t_{j+1} | t_j, \ldots, t_1, B)}$$

Using Bayes' Rule, PLR_n is rewritten as:

$$PLR_n = \frac{\dfrac{p(A | t_{j+n}, \ldots, t_1)\, p(t_{j+n}, \ldots, t_{j+1} | t_j, \ldots, t_1)}{p(A | t_j, \ldots, t_1)}}{\dfrac{p(B | t_{j+n}, \ldots, t_1)\, p(t_{j+n}, \ldots, t_{j+1} | t_j, \ldots, t_1)}{p(B | t_j, \ldots, t_1)}}$$

$$= \frac{p(A | t_{j+n}, \ldots, t_1)}{p(B | t_{j+n}, \ldots, t_1)} \cdot \frac{p(B | t_1, \ldots, t_j)}{p(A | t_1, \ldots, t_j)}$$

If the initial predictions were based only on prior belief, the second factor of the last equation is the prior odds ratio. If the user is indifferent between models A and B at this point, this ratio has a value of 1, since p(A) = p(B) . The last equation is then rewritten as:

$$PLR_n = \frac{w_A}{1 - w_A}$$

This is the posterior odds ratio, where w_A is the posteriOr belief that A is true after making predictions with both A and B and comparing them with actual behavior. If $PLR_n \to \infty$ as $n \to \infty$, model B is discarded in favor of model A.

A model can also be evaluated to determine whether the predictions it makes are biased. One way of doing this is to draw a u-plot. Consider the following transformation:

$$u_i = F_i(t_i)$$

Each u_i is a probability integral transform of the observed time to failure t_i using the previously calculated predictor F_i based upon $t_1, t_2, \ldots, t_{i-1}$. In other words, u_i is the probability that the software will fail before time t_i. If each F_i were identical to the true, but hidden, F_i, the u_i would be realizations of independent identically-distributed (iid) random variables whose values lie in the interval [0, 1]. The closeness of F_i to F_i can be examined to

determine the extent to which the model is biased. One way of drawing this is to draw the cumulative distribution functions (calf) for F_i and F_i and determine the maximum vertical distance (Kolmogorov distance) between them. The value of the Kolmogorov distance measures the extent to which the model is biased. Furthermore, if the cdf for F_i is above that for F_i, the model will yield optimistic (too large) estimates for the time to failure. Otherwise, if the cdf for F_i is below that for F_i, the model's estimates of time to failure will tend to be pessimistic (larger than the observed times to failure) .

Another method of analyzing a model's bias is to form the y-plot . When the cdf of the u_i was plotted in the u-plot, the temporal ordering of the u_i was lost. This can cause a model which is optimistic in the early stages, but pessimistic later on, to appear unbiased when examining the u-plot. To examine the u_i for trend, their temporal ordering must be preserved. This can be done using the following sequence of transformations:

$$x_i = -\ln(1 - U_i)$$

$$y_i = \frac{\sum_{j=1}^{i} x_j}{\sum_{j=1}^{n} x_j}$$

where n is the total number of failures observed. The cdf of the y_i and the cdf for F_i are then drawn, as was done for the u-plot. This y-plot reveals trends in the u_i. The point at which the cdf of the y_i departs from the cdf for F_i indicates the time at which the estimates made by the model are biased.

Finally, a measure of the noisiness of a model's estimates can be made. The median variability is defined as:

$$\sum_{i} \left| \frac{m_i - m_{i-1}}{m_{i-1}} \right|$$

where m_i is the predicted median of the random variable T_i. Comparing this value for two different models can indicate objectively which model is producing the most variable predictions. However, this does not indicate whether the variability reflects the true variability of the actual reliability.

8.2.8.3 Dealing with Evolving Software

All of the models described in the preceding paragraphs assume that the software being tested will not be undergoing significant

changes during the testing cycle. This is not always the case. A software system undergoing test may be simultaneously undergoing development, with changes being made to the existing software or new functionality being added periodically. To accurately model software reliability in this situation, changes made to the software have to be 'taken into account. There are three ways of handling changes to a program under test. These approaches are:

1. Ignore the change.
2. Apply the component configuration change method.
3. Apply the failure time adjustment technique.

Ignoring changes is appropriate when the total volume of changes is small compared to the overall size of the program. In this case, the continual re-estimation of parameters will reflect the fact that some change is in fact occurring.

The component configuration change approach is appropriate for the situation in which a small number of large changes are made to the software, each change resulting from the addition of independent components (e.g. addition of the telemetry gathering and downlinking capability to a spacecraft command and data subsystem). The reliability of each software component is modeled separately. The resulting estimates are then combined into a reliability figure for the overall system.

The failure time adjustment approach is most appropriately used when a program cannot be conveniently divided into separate independent subsystems and the program is changing rapidly enough to produce unacceptable errors in estimating the software's reliability. The three principal assumptions that are made in failure time adjustment are:

1. The program evolves sequentially. At any one time, there is only one path of evolution of the program for which reliability estimates are being made.
2. Changes in the program are due solely to growth. Differences between' version k and version k+1 are due entirely to new code being added to version k.
3. The number of faults introduced by changes to the program are proportional to the volume of new code.

Figure 6 provides an example of a software system to which failure time adjustment techniques could be applied. This figure represents the cumulative number of errors for the system whose failure frequencies are shown in Figures 4 and 5. Recall that. Figures 4 and 5 show abrupt changes in the failure frequency at. week 34. For this example, this change is attributed to the addition of new functionality to the software under test. The testing method remains the same during the two stages.

' When testing proceeds in two stages, the expected number of failures as a function of time will follow a known curve during the first stage (weeks 0-14 in Figure 6). The parameters of this curve will depend on the fault content and the total amount of code being executed in this stage. After the fourteenth week of testing (week 14 in Figure 6, denoted by t' *in* the text), additional code that implements the remainder the system is added. At this point, the curve representing the expected number of errors will switch to the one that would have occurred for a system in its final configuration. The curve, however, is temporally translated, the amount of translation depending on the number of failures that were experienced during the first test stage. The translation can be determined by modeling the first and second stages independently,

If the Goel-Okumoto model is used, the parameters for the first and second stages are as follows:

	a	b
First Stage	317	.0487787
Second Stage	413	.0461496

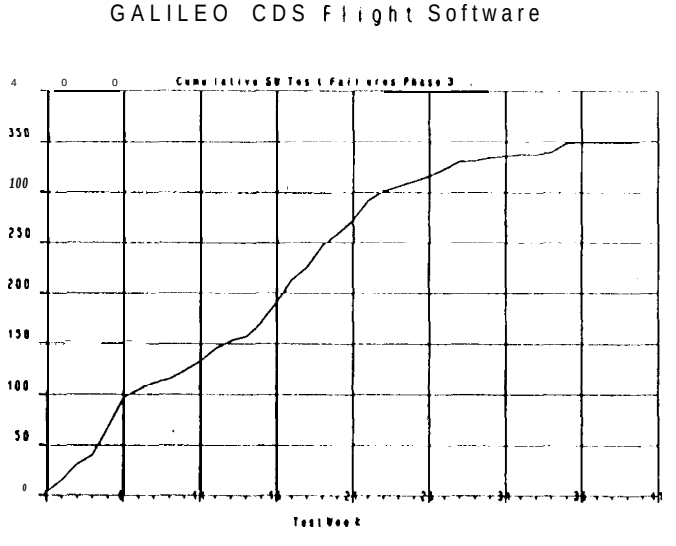

Figure 6 - Cumulative Number of Errors for a Spacecraft Control and Data Subsystem

In the first stage, then, the expected number of failures is given by $\mu_1(t) = a_1(1 - \exp(-bit))$. Substituting the values of a and b in the table above, and using a value of 14 for t, the expected number of errors is 156. Now assume that testing had started using the final configuration, represented by the second stage. The expected number of errors would be given by $\mu_2(t) = a_2(1 - \exp(-b_2t))$, using the values of a and b given for the second stage. For the second stage, the number of failures expected to be observed during the first stage (156) would be observed in 10 time units, denoted by t^*, rather than in 14. Therefore, in going from the first to the second stage of testing, the expected number of failures, $\mu(t)$, will be a translated version of the expression for $\mu_2(t)$. The amount of translation is given by $t' - t'$, which in this case is 4.

$$\mu(t) = a_2(1 - \exp(-b_2(t - (t' - t^*))))$$

or

$$p(t) = a_2(1 - \exp(-b_2(t - 4)))$$

The following general expression denotes the transformation that will take a value of the time to failure t_i to the adjusted value t_i:

$$t_i = Q_{12}(t_i;\ I_1,\ I_2,\ \Delta I_1,\ \Delta I_2,\ \beta_{(1)})$$

where

t_i = the interfailure time t_i, observed in the unchanged system, transformed to the expected interfailure time in the changed system.

I_1 = the total number of executable instructions (developed and inherited) for stage 1..

I_2 = the total number of executable instructions (developed and inherited) for stage 2.

ΔI_1 = the number of executable instructions developed for stage 1.

ΔI_2 = the number of executable instructions developed for stage 2.

$\beta_{(1)}$ = parameters of the model for the first stage of testing.

The specific expression for the transformation depends on the software reliability model that is used. Once the adjusted failure times t_i have been found, these t_i, rather than the unadjusted t_i, are used in making reliability estimates in the future.

The two-stage transformation can be generalized to a testing phase in which there are more than two stages. For two testing

stages k and 1, stage k preceding stage 1, the transformation is written as:

$$t_i = Q_{k1}(t_i;\ I_k,\ I_1,\ \Delta I_k,\ \Delta I_1,\ \beta_{(1)})$$

The parameters of this transformation have the same meaning as for the two-stage transformation. Specific forms of this transformation for the Muss Basic and Muss-Okumoto models are given below. The form for the Muss Basic model is:

$$t_i = -\frac{1}{\phi_k I_k} \ln\left\{1 - \frac{\Delta I_k}{\Delta I_1}\left[1 - e^{-\phi_k t_i}\right]\right\}$$

ϕ_k = the failure intensity decay factor for stage k of the testing effort.

I_k = the total number of executable instructions (developed and inherited) for stage k.

I_1 = the total number of executable instructions (developed and inherited) for stage 1.

ΔI_k = the number of executable instructions developed for stage k.

ΔI_1 = the number of executable instructions developed for stage 1.

For the Muss-Okumoto model, the failure transformation time is:

$$\tilde{t}_i = \frac{1}{\phi_1}\left[(\phi_k t_i + 1)^{\frac{\theta_1}{\theta_k}} - 1\right]$$

θ_k = the failure intensity decay parameter for stage k, and
θ_1 = the failure intensity decay parameter for stage 1. //

$\phi = \lambda_0 \theta$

The relation of ϕ_k to ϕ_1 *or* θ_k to θ_1 cannot be determined using current practice. Further details on failure time adjustment can be found in [MUSA87].

8.3 Experimental Results

8.3.1 Comparison of Individual Models

During the course of the JPL study, we found that it would be possible to apply software reliability models software development efforts at JPL. The major difficulty encountered was collecting

the failure history data described in section 8.2.3, although we did find that many development efforts are already set up to collect this type of information. All that would have to be done is to enforce requirements for using the mechanisms already in place.

We found that there was no one "best" model for the development efforts that were studied. This is consistent with the findings reported in [GHALY86]. The following tables summarize the analysis of model applicability for the JPL efforts. For each development effort, the models applied were evaluated with respect to prequential likelihood, model bias, bias trend, and model noise. Each of these criteria was given equal weighting in the overall. ranking. The abbreviations used in the tables are:

- DU - Duane model
- GO - Goel-Okumoto model
- JM - Jelinski-Moranda model
- LM - Littlewood model
- LV - Littlewood-Verrall model
- MO - Muss-Okumoto model
- PL - Prequential Likelihood

Measure	JM	GO	MO	DU	LM	LV
PL	6	3	2	4	5	1
Bias	5	3	3	2	5	1
Trend	5	3	2	6	4	1
Noise	5	3	2	1	5	4
Overall Rank	6	3	2	4	5	1

Model Rankings for Flight System 1

Measure	JM	GO	MO	DU	LM	LV
PL	2	4	5	6	2	1
Bias	3	3	5	1	6	2
Trend	3	4	6	2	4	1
Noise	3	2	1	5	4	6
Overall Rank	2	3	5	4	6	1

Model Rankings for Flight System 2

Measure	JM	GO	MO	DU	LM	LV
PL	3	2	5	6	2	1
Bias	3	3	1	2	5	6
Trend	3	2	5	6	3	1
Noise	5	4	3	1	5	2
Overall Rank	3	2	3	5	5	1

Model Rankings for Flight Subsystem 1

Measure	JM	GO	MO	DU	LM	LV
PL	3	3	3	1	3	2
Bias	4	4	2	1	2	6
Trend	3	3	3	2	3	1
Noise	1	1	1	5	4	6
Overall Rank	3	3	1	1	5	6

Model Rankings for Flight System 3

Measure	JM	GO	MO	DU	LM	LV
PL	1	4	1	6	1	5
Bias	1	1	1	6	1	5
Trend	1	3	4	5	1	6
Noise	3	2	1	5	4	6
Overall Rank	1	4	2	5	2	5

Model Rankings for Ground System 1

8.3.2 Discussion of Results

From the tables above, it's easy to see that a model that performs well for one development effort may do poorly in another. For instance, the Littlewood-Verrall model performs very well for the first three data sets - in fact, it out-performs all of the other models. However, it comes in last for the remaining two development efforts. This inconsistency is repeated for the other five models, as well. There were no clear differences between the development processes for the flight systems and subsystems, certainly none that would favor the selection of one model over another prior to the start of test. These findings suggest that multiple models be applied to the failure data during the test phases of a development effort, preferably models making different assumptions about the error detection and removal processes. In addition, the models should be "continually evaluated for applicability to the failure data. The model or models ranking highest with respect to the evaluation criteria should then be chosen for use in predicting future reliability. The criteria we suggest are the same as those described in [GHALY86], although other criteria, such as traditional goodness-of-fit tests or the Akaike Information Criterion may also be used.

8.4 Linear Combinations of Model Results

Our other finding was that linear combinations of model results appear to provide more accurate predictions than the individual models themselves. We adopted the following strategy in forming combination models:

1. Identify a basic set of models (the component models). If you can characterize the testing environment for the development effort, select models whose assumptions are closest to the actual testing practices.

2. Select models whose predictive biases tend to cancel each other. As previously described, models can have optimistic *or* pessimistic biases.

3. Separately apply each component model to the data.

4. Apply criteria you've selected to weight the selected component models (e.g. changes in 'the prequential likelihood) and fen" the combination model fo-r the final. predictions. Weights can be either static or dynamically determined.

In general, this approach is expressed as a mixed distribution,

$$\hat{f}_i(t) = \sum_{j=1}^{n} w_i^j \hat{f}_i^j(t)$$

where n represents the number of models, the sum of all of the weights w_i^j is 1, and $f_i^j(t)$ represents the predictive probability density function for the j'th component model, given the i-1 observations of failures have been made.

For the JPL failure data, the combination models often outperformed the other six models that were evaluated with respect to the criteria identified and described in the preceding sections. We experimented with three types of combinations:

1. Statically-weighted combinations.
2. Dynamically-weighted combinations, in which weights are determined by comparing and ranking model results.
3. Dynamically-weighted combinations, in which weights are determined by changes in model evaluation criteria.

These types of combinations are further described in the following paragraphs.

8.4.1 Statically-Weighted Linear combinations

Two types of statically-weighted combinations were formed, The first combination, the Equally-Weighted Linear Combination (ELC) model, was formed by assigning equal weights to the Goel-Okumoto, Muss-Okumoto, and Littlewood-Verral models. Because these weights remain constant throughout the modeling process, this combination is very easy to form if the results from the component models are available. Over the failure data sets that were analyzed, the ELC model performed surprisingly well.

8.4.2 Weight Determination Based on Ranking Model Results

Combination models may produce more accurate results if the weights are dynamically assigned rather than remaining static: throughout the modeling process. One way of dynamically assigning weights is based *on* simply ranking component model results. If a combination model contains "n" components, choose a set of "n" values that can be assigned to the components based on a ranking of model results. One of the combinations that we experimented with was the Median-Weighted Linear Combination (MLC), which was composed of the Goel-Okumoto, Littlewood-Verrall, and Muss-Okumoto models. For each failure, the component models would be run, and the results of the models would then be compared. The models predicting the highest and lowest times to the next failure would then be given weig-hts of 0 in the combination, while the prediction in the middle would be given a weight of 1.

The other combination of this type with which we experimented was the Unequally-Weighted Linear Combination (ULC) model. This was formed with the same components as the MLC model - the only difference was that the component models producing the highest and lowest predictions of the time to the next failure were given weights of 1/6 rather than 0, while the component making the middle prediction was given a weight of 4/6.

This type of combination model is not quite as easy to form as the ELC model previously described, but does not require the more complicated calculations required for the last type of combination with which we experimented.

8.4.3 Weight Determination Based on Changes in Prequential Likelihood

The last type of combination with which we experimented was one in which weights were both dynamically determined and assigned. The basis for determining and assigning weights was changes in the prequential likelihood (see chapter TBD) over a small number of observations. There are two ways in which weights can be computed for each model. First, we can look at changes in the prequential likelihood every N observations, and recompute the weights for each combination after every Nth observation. The second way is to recompute weights after every observation, using the changes in the prequential likelihood *over* the most recent N observations to compute and assign the weights. We refer to the first type of combination as a DLC/F/N, which stands for Dynamic Linear Combination with a Fixed window of N observations. The second type of combination is called a DLC/S/N, in which S refers to the sliding window, N observations wide, which is used to recompute the weights.

8.4.4 Discussion of Results

We found that over the JPL failure data sets that we analyzed, as well as for historical failure data reported in [MUSA80], the combination models consistently outperformed the other models that we evaluated. The results are given in the tables on the next page. For the combinations with we worked, which were formed using the Goel-Okumoto, the Littlewood-Verrall, and the Muss-Okumoto models, the combination models sometimes outperformed all of their component models, and never performed worse than the worst component model. The ELC and DLC models performed more consistently than the models. We believe that the ELC model's performance is due to the equal weighting. By contrast, the weighting schemes for the ULC and MLC models may allow weight assignments which do reflect how close the component model results are to one another. Finally, the superior performance of the DLC model is because the weights for each component are dynamically

determined, and are closely related to the likelihood of one component producing more acc-urate predictions than another.

Model / Data	JM	GO	MO	DU	LM	LV	ELC	ULC	MLC	DLC
Muss data set 1	10	9	1	6	8	6	4	2	3	5
Muss data set 2	9	10	6	7	8	1	4	5	2	2
Muss data set 3	6	8	4	9	9	6	4	3	2	1
JPL flight system 1	100	7	6	7	9	2	2	4	5	1
JPL flight system 2	5	7	10	6	9	4	1	3	8	2
JPL flt subsystem 1	8	6	6	8	10	1	1	1	4	5
JPL flight system 3	5	5	8	1	9	10	1	5	4	3
JPL ground system 1	1	5	1	9	3	10	8	7	3	6
Overall rank	8	9	6	7	10	5	1	3	4	1

Summary of model rar ings using all four criteria

Model / Data	JM	GO	MO	DU	LM	LV	ELC	ULC	MLC	DLC
Muss data set 1	10	9	2	8	6	7	5	4	3	1
Muss data set 2	7	9	4	10	7	1	4	4	3	2
Musa data set 3	4	7	4	10	8	9	2	2	4	1
JPL flight system 1	10	7	6	8	9	2	3	4	5	1
JPL flight system 2	5	7	9	10	5	4	2	3	8	1
JPL flt subsystem 1	6	5	8	10	6	2	3	4	8	1
JPL flight system 3	6	6	6	2	6	5	3	4	6	1
JPL ground system 1	2	6	2	10	2	9	8	7	2	1
Overall rank	8	9	6	10	7	4	2	3	4	1

Summary of model rankings using only prec ient: l l: elih od

8.5 Chapter Summary

Problems (10-20 questions to *review* the chapter)

Further Reading:

[GHALY86] A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood: "Evaluation of Competing Software Reliability Predictions," IEEE Transactions on Software Engineering; vol. SE-12, pp. 950-967; Sep. 1986.

[JPL91] "Handbook for Software Reliability Measurement", version 1.0, JPL Publication JPL-D-8672, June 20, 1991.

[LYU91] M. Lyu, "Measuring Reliability of Embedded Software: An Empirical Study with JPL Project Data," published in the Proceedings of the International Conference on Probabilistic Safety Assessment and Management; February 4-6, 1991, Los Angeles, CA.

[LYU91a] M. Lyu and A. Nikora, "A Heuristic Approach for Software Reliability Prediction: The Equally-Weighted Linear Combination Model," published in the proceedings of the IEEE International Symposium on Software Reliability Engineering, May 17-18, 1991, Austin, TX.

[LYU91b] M. R. Lyu, A. Nikora, "Software Reliability Measurements Through Combination Models: Approaches, Results, and a Case Tool," in Proceedings of the 15th Annual International Computer Software and Applications Conference (COMPSAC91), Tokyo, Japan, September, 1991.

[LYU92] M. Lyu and A. Nikora, "Applying Reliability Models More Effectively", IEEE Software, vol. 9, no. 4, pp 43-52, July, 1992

[MUSA80] J. Muss, "Software Reliability Data", tech. report, Rome Air Development Center, Griffis AFB, 1980.

[MUSA87] John D. Muss., Anthony Iannino, Kazuhiro Okumoto, Software Reliability: Measurement, Prediction, Application; McGraw-Hill, 1987; ISBN 0-07-044093-X.

[RADC87] "Methodology for Software Reliability Prediction and Assessment," Rome Air Development Center (RADC)

Technical Report RADC-TR-87-171. volumes 1 and 2, 1987.

Acknowledgements:

The research presented in this chapter was done at the University of Iowa under a faculty starting fund, at Bellcore, and at the Jet Propulsion Laboratory, California Institute of Technology, under a NASA contract through the Director's Discretionary Fund.